The Curry-Howard isomorphism: of proofs and programs

Kazuo Thow

June 3, 2011

The λ-calculus is the assembly language of mathematics.

– Matt Might [1]

Contents

[1] http://matt.might.net/articles/compiling-up-to-lambda-calculus/

1 Introduction

Among the under-appreciated achievements of 20^{th} century mathematics is the precise unification of the notions of *proving* and *computing*. A series of results, primarily by Haskell Curry and William Howard, culminating in a 1969 manuscript [1], exposed a deep connection between the systems known as *intuitionistic natural deduction* and *simply typed λ-calculus*. These are models of logical inference and of computation, respectively.

Following the first four chapters of Sørensen and Urzyczyn's *Lectures on the Curry-Howard Isomorphism* [2] in condensed fashion, we will make this connection precise. We begin by introducing a simple formal system, the bare λ-calculus without types and show how, surprisingly enough, it meets our need for a complete model of computation. Next we lay the foundations for a system of constructive proof: natural deduction with intuitionistic logic. To solidify the notions of *domain* and *range* for computable functions we strengthen our model of computation into the simply typed λ-calculus, a system ripe with connections to natural deduction.

Put in slightly poetic but not inaccurate terms, the Curry-Howard isomorphism says that *a program does what its corresponding proof says*. And, conversely, *a proof says what its corresponding program does*.

2 Type-free λ-calculus

2.1 A simple model of computation

To obtain a complete model of computation, only a surprisingly simple formalism is required. Beginning with the formalism itself, the untyped λ-calculus, we will see how its structure can encode the notion of a function, an application/evaluation, arithmetic and even recursion.

First we define the set Λ of *λ-terms* by the following grammar:

$$\begin{aligned} \Lambda &::= V \mid (\lambda V \Lambda) \mid (\Lambda\Lambda) \\ V &::= v_0 \mid v_1 \mid v_2 \mid \cdots, \end{aligned}$$

where the v_i are *variables*, terminals in the above grammar. Hence a λ-term is either a variable reference, an abstraction (a parametrization of a body term, denoted by the symbol λ followed by the parameter variable), or an application (of one λ-term to another). Function application takes highest precedence, and by convention associates to the left; $PQR = (PQ)R$ for $P, Q, R \in \Lambda$. Some examples of λ-terms generated by this grammar are

$$v_0, \quad \lambda v_0.v_0, \quad \lambda v_0.\lambda v_1.(v_0\, v_1) \quad \in \Lambda,$$

where a period (.) is often used as a notational shorthand to avoid writing parentheses around the body of a term. The first of these examples is simply a reference to the variable v_0, the second can be thought of as the identity function, and the third a λ-term which applies its first parameter to its second.

To provide a meaning for the notion of "variables upon which a λ-term depends", we have the set $FV(M) \subseteq V$ of *free variables* of a λ-term M:

$$\begin{aligned} FV(x) &= \{x\}, \\ FV(\lambda x.P) &= FV(P) \setminus \{x\}, \\ FV(P\,Q) &= FV(P) \cup FV(Q), \end{aligned}$$

where $x \in V$. If $FV(M) = \{\}$, we say M is *closed*. For example,

$$(\lambda x.x)v_0, \quad \text{and} \quad \lambda x.x\, y$$

are closed and open, respectively, with the free variables of the latter consisting of y.

(Strictly speaking the above three restrictions on FV define a *set* of functions mapping $\Lambda \to \mathcal{P}(V)$; it can be seen easily that there is a unique FV satisfying the three rules, and so we refer simply to *the* [unique] set of free variables of a λ-term.)

Immediately we must also define a notion of *substitution*: the substitution $M[x := N]$ of N for x in the λ-term M is defined by

$$\begin{aligned}
x[x := N] &= N, \\
y[x := N] &= y, \\
(P\,Q)[x := N] &= P[x := N]\,Q[x := N], \\
(\lambda x.P)[x := N] &= \lambda x.P \quad \text{(parameter names are arbitrary)} \\
(\lambda y.P)[x := N] &= \lambda y.P[x := N], && \text{if } y \notin FV(N) \text{ or } x \notin FV(P), \\
(\lambda y.P)[x := N] &= \lambda z.P[y := z][x := N], && \text{if } y \in FV(N) \text{ and } x \in FV(P),
\end{aligned}$$

where z is chosen to be the lexically first variable in V not in $FV(P) \cup FV(N)$.

We can now introduce a relation on Λ to specify a notion of simplification, or reduction, of one λ-term to another. Let $\to_\beta$ be the smallest binary relation on Λ so that

$$(\lambda x.P)Q \to_\beta P[x := Q],$$

and if $P \to_\beta P'$, then

$$\begin{aligned}
&\forall x \in V : \lambda x.P \to_\beta \lambda x.P', \quad \text{(abstraction is preserved)} \\
&\forall Z \in \Lambda : P\,Z \to_\beta P'\,Z, \quad \text{(behavior as a function is preserved)} \\
&\forall Z \in \Lambda : Z\,P \to_\beta Z\,P' \quad \text{(behavior as an argument is preserved).}
\end{aligned}$$

We take $\to_\beta^*$ (the multi-step β-reduction, or simply *β-reduction*) to be the transitive reflexive closure of $\to_\beta$. For example,

$$\begin{aligned}
(\lambda f.\lambda a.f\,a)(\lambda e.e)x &\to_\beta (\lambda a.(\lambda e.e)a)x \\
&\to_\beta (\lambda e.e)x \\
&\to_\beta x,
\end{aligned}$$

so $(\lambda f.\lambda a.f\,a)(\lambda e.e)x \to_\beta^* x$. We say the *$\beta$-normal form* of a term M is the term N for which no further β-reductions $N \to_\beta P$ are possible.

β-equality is denoted by $A =_\beta B$ when $A \to_\beta^* B$ and $B \to_\beta^* A$. That is, $=_\beta$ is the symmetric closure of the relation $\to_\beta^*$.

Now we are in a position to see one troubling consequence of our model's type-freeness: we can define a function $\omega = \lambda x.(x\,x)$ which applies its parameter to itself. What happens when we attempt to find the β-normal form of the term $\omega\,\omega$?

$$\omega\,\omega = (\lambda x.(x\,x))\,(\lambda x.(x\,x)) \to_\beta (\lambda x.(x\,x))\,(\lambda x.(x\,x)) \to_\beta \ldots$$

The recursive reduction steps do not reach the base cases defined by our substitution rules above; the process of β-reducing $\omega\,\omega$ never ends. To develop the Curry-Howard isomorphism we will need, among other things, a guarantee that the "normal form" of any term is reachable in a finite number of reduction steps. This issue will be resolved when we come to the simply typed λ-calculus and its weak normalization property.

2.2 β-reduction as computation

2.2.1 Church-Rosser theorem

The **Church-Rosser theorem** states that the relation $\to_\beta^*$ satisfies the *diamond property*; for $M_1, M_2, M_3 \in \Lambda$, if $M_1 \to_\beta^* M_2$ and $M_1 \to_\beta^* M_3$, then there exists $M_4 \in \Lambda$ such that $M_2 \to_\beta^* M_4$ and $M_3 \to_\beta^* M_4$. This allows us to speak of the *β-normal form* of a λ-term M; we can uniquely identify an N such that $M \to_\beta^* N$ and N has no further β-reduction.

This is analogous to how, for instance, the arithemtic expressions $(4 + 2) \cdot 2$ and $6 \cdot (5 - 3)$ both have the same value 12 when fully reduced. Importantly, the normal form obtained is independent of the particular sequence of reduction steps, so long as the sequence brings us to a form which cannot be reduced further.

2.2.2 Expressing arithmetic

Encoding natural numbers $n \in \mathbb{N}$ as λ-terms is simple, if somewhat non-obvious. First, some notation: for $F, A \in \Lambda$ let $F^n(A)$ denote the n-fold application of F to A so that $F^0(A) = A$, $F^2(A) = F(F(A))$, and so on. Then the nth *Church numeral* is defined as

$$c_n = \lambda s.\lambda z.s^n(z),$$

and we can express the successor function $n \mapsto n + 1$ by

$$S = \lambda x.\lambda s.\lambda z.s\,(x\,s\,z).$$

To see that applying S to a Church numeral has the effect of adding one to the integer it represents, simply β-reduce:

$$\begin{aligned} S\,c_n &= (\lambda x.\lambda s.\lambda z.s\,(x\,s\,z))\,(\lambda s.\lambda z.s^n(z)) \\ &\to_\beta \lambda s.\lambda z.s((\lambda s.\lambda z.s^n(z))\,s\,z) \\ &\to_\beta \lambda s.\lambda z.s((\lambda z.s^n(z))\,z) \\ &\to_\beta \lambda s.\lambda z.s(s^n(z)) \\ &= c_{n+1}. \end{aligned}$$

2.2.3 Expressing conditionals

Type-free λ-calculus can also encode the notion of conditional execution, a basic and highly useful construct in any modern programming language. We define

$$\begin{aligned} &\mathbf{true} = \lambda x.\lambda y.\,x, \\ &\mathbf{false} = \lambda x.\lambda y.\,y, \\ &\mathbf{if}\,B\,\mathbf{then}\,P\,\mathbf{else}\,Q = B\,P\,Q. \end{aligned}$$

(Recall that application of λ-terms associates to the left.) Then

$$\begin{aligned} \mathbf{if\,true\,then}\,P\,\mathbf{else}\,Q &\to_\beta (\lambda x.\lambda y.\,x)\,P\,Q \\ &\to_\beta (\lambda y.P)\,Q \\ &\to_\beta P, \end{aligned}$$

so our conditional construct behaves as expected – the case for $B = \mathbf{false}$ is similar.

2.3 λ-definability

What we mean precisely by a λ-term F "expressing" a numeric function $f : \mathbb{N}^k \to \mathbb{N}$ is that

$$F\,c_{n_1}\,\ldots\,c_{n_k} =_\beta c_{f(n_1,\ldots,n_k)},$$

for all $n_1, \ldots, n_k \in \mathbb{N}$, where the c_j are Church numerals, as before. We then say that the numeric function f is λ-*definable* and defined by F.

The class of *recursive functions* is of central interest in computability theory, and it is defined as follows:

The *initial functions* are recursive:

(i) *Zero*: $Z(n) = 0$,

(ii) *Successor*: $S^+(n) = n + 1$,

(iii) *Projections*: $U_i^k(n_1, \ldots, n_k) = n_i$, for all $1 \le i \le k$.

The set of recursive functions is taken to be minimally closed under *composition*, *primitive recursion* and *minimization*:

(1) *Composition*: If $g : \mathbb{N}^k \to \mathbb{N}$ and $h_1, \ldots, h_k : \mathbb{N}^m \to \mathbb{N}$ are recursive, then $f : \mathbb{N}^m \to \mathbb{N}$ is recursive;

$$f(n_1, \ldots, n_m) = g(h_1(n_1, \ldots, n_m), \ldots, h_k(n_1, \ldots, n_m)).$$

(2) *Primitive recursion*: If $g : \mathbb{N}^m \to \mathbb{N}$ and $h : \mathbb{N}^{m+2} \to \mathbb{N}$ are recursive, then $f : \mathbb{N}^{m+1} \to \mathbb{N}$ is recursive;

$$f(0, n_1, \ldots, n_m) = g(n_1, \ldots, n_m),$$
$$f(n+1, n_1, \ldots, n_m) = h(f(n, n_1, \ldots, n_m), n, n_1, \ldots, n_m).$$

(3) *Minimization*: If $g : \mathbb{N}^{m+1} \to \mathbb{N}$ is recursive and for all $n_1, \ldots, n_m$ we have $g(n, n_1, \ldots, n_m) = 0$ for some n, then $f : \mathbb{N}^m \to \mathbb{N}$ is recursive;

$$f(n_1, \ldots, n_m) = \min_n \{g(n, n_1, \ldots, n_m) = 0\}.$$

It is easily seen that the initial functions are λ-definable. The zero and successor functions have been established by previous discussion, and for the k-ary projection function let $U_i^k = \lambda v_1 \ldots \lambda v_k . v_i$, a simple generalization of the "selector" terms defining **true** and **false**.

LEMMA: (Closure of λ-definability under composition) If $G \in \Lambda$ defines $g : \mathbb{N}^k \to \mathbb{N}$ and $H_1, \ldots, H_k \in \Lambda$ define $h_1, \ldots, h_k : \mathbb{N}^m \to \mathbb{N}$, then $f : \mathbb{N}^m \to \mathbb{N}$ (as in the definition of composition above) is λ-definable and defined by

$$F = \lambda x_1 \ldots \lambda x_m . G\,(H_1\, x_1 \ldots x_m) \ldots (H_k\, x_1 \ldots x_m).$$

PROOF:

$$\begin{aligned} F\, c_{n_1} \ldots c_{n_m} &\to_\beta G(H_1\, c_{n_1} \ldots c_{n_m}) \ldots (H_k\, c_{n_1} \ldots c_{n_m}) \\ &= G\, c_{h_1(n_1,\ldots,n_m)} \ldots c_{h_k(n_1,\ldots,n_m)} && \text{(by definability of } h_1, \ldots, h_k) \\ &= c_{g(h_1(n_1,\ldots n_m),\ldots,h_k(n_1,\ldots,n_m))} && \text{(by definability of } g) \\ &= c_{f(n_1,\ldots,n_m)} && \text{(by definition of } f). \quad \square \end{aligned}$$

The proofs of closure under primitive recursion and minimization are similar. Taking these closure properties together, we have the following result due to Kleene:

Theorem: *All recursive functions are λ-definable.*

We have thus established the power of type-free λ-calculus as a model of computation.

3 Intuitionistic propositional logic

Define the set of *formulas* Φ over a countable infinite set P of *propositional variables* by the following grammar:

$$\Phi ::= \bot \mid P \mid (\Phi \to \Phi) \mid (\Phi \wedge \Phi) \mid (\Phi \vee \Phi).$$

So our connectives, familiar from classical logic, are implication ($\to$), disjunction ($\wedge$) and conjunction ($\vee$). A *context* Γ is a finite subset of Φ; at this point we tentatively introduce the notation $\Gamma \vdash \varphi$ to, in intuitive terms, denote that the formula φ is a logical consequence of the formulas in Γ under the usual derivation rules of classical logic.

The *implicational fragment* of the intuitionistic propositional calculus (IPC) is that part IPC($\to$) of IPC consisting of the reduced grammar $\Phi ::= \bot \mid P \mid (\Phi \to \Phi)$ and the introduction ($\to I$) and elimination ($\to E$) rules for implication, and the axiom scheme (Ax):

$$\Gamma, \varphi \vdash \varphi\,(\text{Ax}), \qquad \frac{\Gamma, \varphi \vdash \psi}{\Gamma \vdash \varphi \to \psi}\ (\to I), \qquad \frac{\Gamma \vdash \varphi \to \psi \qquad \Gamma \vdash \varphi}{\Gamma \vdash \psi}\ (\to E),$$

where assumptions are written above the horizontal bar and conclusions below it.

The following lemma establishes the sufficiency of IPC($\to$):

LEMMA: *If $\Gamma \vdash \varphi$ can be derived in IPC then it can be derived in IPC($\to$).*

Hence we will focus on IPC($\to$) as our model of logical inference.

4 λ-calculus with simple types: the Church system

While the type-free λ-calculus provides us with a powerful model of computation, the sense in which a non-closed λ-term can be considered a *function*, with domain and range, was not at all specified. Indeed, any non-closed λ-term can take any λ-term as its parameter.

To resolve this issue we introduce the notion of *simple types* over a set U of *type variables*:

$$\Pi ::= U \mid \Pi \to \Pi.$$

A *context* Γ is a set $\{x_1 : \tau_1, \ldots, x_n : \tau_n\}$ of *variable* : *type* pairs. Denote by $\mathrm{dom}(\Gamma)$ the *domain* $\{x_1, \ldots, x_n\}$ of the context Γ and write $|\Gamma|$ for its *range* $\{\tau \in \Pi \mid (x : \tau) \in \Gamma, \text{ for some } x\}$.

We describe the set of simply typed λ-terms by the grammar

$$\Lambda_\Pi ::= V \mid (\lambda x : \Pi.\, \Lambda_\Pi) \mid (\Lambda_\Pi\, \Lambda_\Pi).$$

An important note to make about the restrictions already imposed by this slight modification to the original λ-calculus is that Λ_Π is a strict subset of Λ; some terms in the untyped λ-calculus do not appear in the simply typed calculus. For instance, we cannot assign a consistent typing to the free variables of $\omega = \lambda x.(x\, x)$.

The *typeability relation* $\vdash$ provides further structure: we define the relation $\vdash$ on $C \times \Lambda_\Pi \times \Pi$ (where C is the set of all contexts) by

$$\frac{}{\Gamma, x : \tau \vdash x : \tau,} \qquad \frac{\Gamma, x : \sigma \vdash M : \tau}{\Gamma \vdash \lambda x : \sigma.M \ : \ \sigma \to \tau,} \qquad \frac{\Gamma \vdash M \ : \ \sigma \to \tau \qquad \Gamma \vdash N : \sigma}{\Gamma \vdash M\, N \ : \ \tau},$$

where $x \notin \mathrm{dom}(\Gamma)$ for the first and second rules. These three rules correspond to variable refrence, abstraction and application, respectively. "$\Gamma \vdash M : \sigma$" means that the term M has type σ in the context Γ.

With substitution and β-reduction defined very similarly (with care taken to account for types), the diamond property also holds for this system just as it did for untyped λ-calculus.

4.1 Uniqueness of types

One of the key properties of typed λ-calculus is that terms have unique types:

Lemma: *Suppose* $M \in \Lambda_\Pi$. *If* $\Gamma \vdash M : \sigma$, $\Gamma \vdash N : \tau$, *and* $M =_\beta N$ *then* $\sigma = \tau$

Proof: First note that if $M = N$ then by a simple induction on M, if M can be identified with type σ and can be identified with type τ, then $\sigma = \tau$. In the general case where $M =_\beta N$ we have the Church-Rosser property: $M \to_\beta L$ and $N \to_\beta L$ for some $L \in \Lambda_\Pi$. This reduces to the simpler case just mentioned, and so $\sigma = \tau$.

4.2 Weak normalization

Adding simple types to our model of computation brings another crucial advantage: avoidance of never-ending reduction chains as in the $\omega\,\omega \to_\beta \omega\, omega \to_\beta \cdots$ example given previously. (For brevity we omit Sørensen and Urzyczyn's proof of this fact, which uses the notion of *height* of a simply typed term – essentially the level of $\to$-nesting in its type.)

Theorem: (Weak normalization of Λ_Π) *If* $\Gamma \vdash M : \sigma$, *then there is a finite reduction sequence* $M \to_\beta M_1 \to_\beta \cdots \to_\beta M_n$ *with* M_n *a normal form.*

5 Formulas as types

Development of the foregoing machinery of λ-calculus and natural deduction now allows us to state and prove the connection between these two systems. Inuitively, having a proof of $\Gamma \vdash \varphi$ is equivalent to having a program that constructs an object of type φ from the objects in Γ.

(In the following we implicitly make use of the fact that when the set P of propositional variables is equal to the set U of type variables, then Φ, propositional formulas in IPC($\rightarrow$), is the same as Π, the set of simple types.)

Theorem: (Curry-Howard isomorphism)

(1) *If* $\Gamma \vdash M : \varphi$, *then* $|\Gamma| \vdash \varphi$.

(2) *If* $\Gamma \vdash \varphi$ *then there exists* $M \in \Lambda_\Pi$ *such that* $\Delta \vdash M : \varphi$, *where* Δ *is the context whose range consists of formulas (types) in* Γ.

Proof: (1) follows simply by induction on derivations of $\Gamma \vdash M : \varphi$. For (2) we proceed by induction on the derivation of $\Gamma \vdash \varphi$; let $\Delta = \{x_\varphi : \varphi \,|\, \varphi \in \Gamma\}$ and consider the three cases arising from minimal IPC:

Suppose the derivation is simply $\Gamma, \varphi \vdash \varphi$. Then if $\varphi \in \Gamma$ we have $\Delta \vdash x_\varphi : \varphi$. Otherwise, if $\varphi \notin \Gamma$ then the context Δ together with a typing for φ will do; $\Delta, x_\varphi : \varphi \vdash x_\varphi : \varphi$.

Next suppose the derivation ends with $\rightarrow$-elimination – that is, the root of the proof tree and its two children are

$$\frac{\Gamma \vdash \varphi \rightarrow \psi \qquad \Gamma \vdash \varphi}{\Gamma \vdash \psi}.$$

We have by the inductive hypothesis that $\varphi \rightarrow \psi$ and ϕ are typeable; $\Delta \vdash M : \varphi \rightarrow \psi$ and $\Delta \vdash N : \varphi$. So by composition of types, $\Delta \vdash M\,N : \psi$.

Finally, suppose the derivation ends in $\rightarrow$-introduction:

$$\frac{\Gamma, \varphi \vdash \psi}{\Gamma \vdash \varphi \rightarrow \psi.}$$

If $\varphi \in \Gamma$: From the induction hypothesis we have that $\Delta \vdash M : \psi$. Adding to the context Δ will not affect the typeability of M, so for $x \notin \mathrm{dom}(\Delta)$, $\Delta, x : \varphi \vdash M : \psi$. Then by the abstraction rule in the rules defining $\vdash$ we have $\Delta \vdash \lambda x : \varphi.M : \varphi \rightarrow \psi$. If $\varphi \notin \Gamma$: By the induction hypothesis, $\Delta, x_\varphi : \varphi \vdash M : \psi$. Then it immediately follows that $\Delta \vdash \lambda x_\varphi : \varphi.M : \varphi \rightarrow \psi$. $\square$

6 Conclusions and implications

In short, we have shown that two simple models for apparently quite different domains (natural deduction and computation of recursive functions) share both a deep syntactic and semantic connection. The formula produced by a proof in natural deduction is the type of a simply typed λ-term; this term in turn has a proof corresponding to its type.

One of the stronger theoretical implications of this version of the Curry-Howard isomorphism, briefly discussed by Sørensen and Urzyczyn [2], is that *proof normalization* (the process of eliminating redundant steps from constructive proofs in natural deduction) and β-reduction turn out to be two sides of the same coin. Intuitively this can be grasped by thinking of proof normalization as a conclusion-preserving operation and, on the computational side, β-reduction as a type-preserving operation.

Eminently practical applications of the Curry-Howard isomorphism are only beginning to be exploited. Many functional programming languages – those languages built around some form of the λ-calculus for their model of computation – including Haskell and ML possess type systems based on Curry's formulation of the simple types, as opposed to the Church formulation briefly presented here. Curry's formulation lacks the uniqueness of types property carried by the Church system, and so applied type theory research has directed some of its attention to the problem of making type systems for such languages expressive enough that guarantees about program behavior can be encoded in the program's internal types. Sheard [3], for example, has studied Generalized Abstract Data Types in the context of Haskell to create language constructs for expressing such guarantees as the fact that lists produced by a sorting function will in fact be sorted, along with much more nontrivial properties.

References

[1] W. A. Howard. The formulae-as-types notion of construction. 1980.

[2] Morten Heine B. Sørensen and Pawel Urzyczyn. Lectures on the curry-howard isomorphism. 2006.

[3] Tim Sheard. Putting curry-howard to work. In *In Proceedings of ACM Workshop on*, pages 74–85. ACM, 2005.